ISO TC184/SC4/WG11 N066

Date: December18, 1998 **Supersedes WG11 N052**

PRODUCT DATA REPRESENTATION AND EXCHANGE

Part: 1? **Title: EXPRESS-X Language Reference Manual**

Purpose of this document as it relates to the target document is:

x Primary content
Issue discussion
Alternate proposal
Partial content

Current status: Working Draft
Version

ABSTRACT:
Draft of EXPRESS-X language reference manual.

KEYWORDS
EXPRESS
EXPRESS-X
Mapping language

Document status/dates (dd/mm/yy)

Part Documents		**Other SC4 Documents**	
X	Working draft	_____	Working
_____	Project draft	_____	Released
_____	Released draft	_____	Confirmed
_____	Technically complete	_____	Released
_____	Editorially complete		
_____	ISO Committee Draft		
_____			

Project leader: Martin Hardwick
Address: STEP Tools, Inc.
1223 Peoples Avenue
Troy, NY 12180
USA
Telephone: +1 (518) 276-2848
Fax: +1 (518) 276-8471
E-mail: hardwick@steptools.com

Editor: Peter Denno
Address: NIST
Quince Orchard Road,
Gaithersburg, MD 20899
USA
Telephone: +1 (301) 975-3595
Fax: +1 (301) 975-4694
E-mail: peter.denno@nist.gov

Comments to Reader:

This draft remains technically and editorially incomplete. It is being distributed to elicit feedback from interested parties and to allow early prototype implementation to begin. Potential implementers should be aware of the early and volatile nature of the language.

Contents

Foreword

The International Organization for Standardization (ISO) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

International Standard ISO 10303-1**???** was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data*.

ISO 10303 consists of the following parts under the general title Industrial automation systems and integration - Product data representation and exchange:

- Part 1, Overview and fundamental principles;
- Part 11, Description methods: The EXPRESS language reference manual;
- Part 12, Description methods: The EXPRESS-I language reference manual;
- Part 21, Implementation methods: Clear text encoding of the exchange structure;
- Part 22, Implementation methods: Standard data access interface specification;
- Part 23, Implementation methods: C++ language binding to the standard data access interface;
- Part 24, Implementation methods: C language binding to the standard data access interface;
- Part 26, Implementation methods: Interface definition language binding to the standard data access;
- Part 31, Conformance testing methodology and framework: General concepts;
- Part 32, Conformance testing methodology and framework: Requirements on testing laboratories and clients;
- Part 33, Conformance testing methodology and framework: Structure and use of abstract test suites;
- Part 34, Conformance testing methodology and framework: Abstract test methods;

- Part 35, Conformance testing methodology and framework: Abstract test methods for SDAI implementations;
- Part 41, Integrated generic resource: Fundamentals of product description and support;
- Part 42, Integrated generic resources: Geometric and topological representation;
- Part 43, Integrated generic resources: Representation structures;
- Part 44, Integrated generic resources: Product structure configuration;
- Part 45, Integrated generic resources: Materials;
- Part 46, Integrated generic resources: Visual presentation;
- Part 47, Integrated generic resources: Shape variation tolerances;
- Part 49, Integrated generic resources: Process structure and properties;
- Part 101, Integrated application resources: Draughting;
- Part 104, Integrated application resources: Finite element analysis;
- Part 105, Integrated application resources: Kinematics;
- Part 106, Integrated application resources: Building construction core model;
- Part 201, Application protocol: Explicit draughting;
- Part 202, Application protocol: Associative draughting;
- Part 203, Application protocol: Configuration controlled design;
- Part 204, Application protocol: Mechanical design using boundary representation;
- Part 205, Application protocol: Mechanical design using surface representation;
- Part 207, Application protocol: Sheet metal die planning and design;
- Part 208, Application protocol: Life cycle management - Change process;
- Part 209, Application protocol: Composite and metallic structural analysis and related design;
- Part 210, Application protocol: Electronic assembly, interconnect, and packaging design;
- Part 212, Application protocol: Electrotechnical design and installation
- Part 213, Application protocol: Numerical control process plans for machined parts;

- Part 214, Application protocol: Core data for automotive design;
- Part 215, Application protocol: Ship arrangement;
- Part 216, Application protocol: Ship moulded forms;
- Part 217, Application protocol: Ship piping;
- Part 218, Application protocol: Ship structures;
- Part 220, Application protocol: Process planning, manufacture, and assembly of layered electronic products;
- Part 221, Application protocol: Functional data and their schematic representation for process plant;
- Part 222, Application protocol: Exchange of product data for composite structures;
- Part 223, Application protocol: Exchange of design and manufacturing product information for casting parts;
- Part 224, Application protocol: Mechanical product definition for process plans using machining features;
- Part 225, Application protocol: Building elements using explicit shape representation;
- Part 226, Application protocol: Ship mechanical systems;
- Part 227, Application protocol: Plant spatial configuration;
- Part 228, Application protocol: Building services: Heating, ventilation, and air conditioning;
- Part 229, Application protocol: Exchange of design and manufacturing product information for forged parts;
- Part 230, Application protocol: Building structural frame: Steelwork;
- Part 231, Application protocol: Process engineering data: Process design and process specification of major equipment;
- Part 232, Application protocol: Technical data packaging core information and exchange;
- Part 301, Abstract test suite: Explicit draughting;
- Part 302, Abstract test suite: Associative draughting;
- Part 303, Abstract test suite: Configuration controlled design;

- Part 304, Abstract test suite: Mechanical design using boundary representation;
- Part 305, Abstract test suite: Mechanical design using surface representation;
- Part 307, Abstract test suite: Sheet metal die planning and design;
- Part 308, Abstract test suite: Life cycle management - Change process;
- Part 309, Abstract test suite: Composite and metallic structural analysis and related design;
- Part 310, Abstract test suite: Electronic assembly, interconnect, and packaging design;
- Part 312, Abstract test suite: Electrotechnical design and installation;
- Part 313, Abstract test suite: Numerical control process plans for machined parts;
- Part 314, Abstract test suite: Core data for automotive mechanical design;
- Part 315, Abstract test suite: Ship arrangement;
- Part 316, Abstract test suite: Ship moulded forms;
- Part 317, Abstract test suite: Ship piping;
- Part 318, Abstract test suite: Ship structures;
- Part 320, Abstract test suite: Process planning, manufacture, and assembly of layered electronic products;
- Part 321, Abstract test suite: Functional data and their schematic representation for process plant;
- Part 322, Abstract test suite: Exchange of product data for composite structures;
- Part 323, Abstract test suite: Exchange of design and manufacturing product information for casting parts;
- Part 324, Abstract test suite: Mechanical product definition for process plans using machining features;
- Part 325, Abstract test suite: Building elements using explicit shape representation;
- Part 326, Abstract test suite: Ship mechanical systems;
- Part 327, Abstract test suite: Plant spatial configuration;
- Part 328, Abstract test suite: Building services: Heating, ventilation, and air conditioning;
- Part 329, Abstract test suite: Exchange of design and manufacturing product information for

forged parts;

– Part 330, Abstract test suite: Building structural frame: Steelwork;

– Part 331, Abstract test suite: Process engineering data: Process design and process specification of major equipment;

– Part 332, Abstract test suite: Technical data packaging core information and exchange;

– Part 501, Application interpreted construct: Edge-based wireframe;

– Part 502, Application interpreted construct: Shell-based wireframe;

– Part 503, Application interpreted construct: Geometrically bounded 2D wireframe;

– Part 504, Application interpreted construct: Draughting annotation;

– Part 505, Application interpreted construct: Drawing structure and administration;

– Part 506, Application interpreted construct: Draughting elements;

– Part 507, Application interpreted construct: Geometrically bounded surface;

– Part 508, Application interpreted construct: Non-manifold surface;

– Part 509, Application interpreted construct: Manifold surface;

– Part 510, Application interpreted construct: Geometrically bounded wireframe;

– Part 511, Application interpreted construct: Topologically bounded surface;

– Part 512, Application interpreted construct: Faceted boundary representation;

– Part 513, Application interpreted construct: Elementary boundary representation;

– Part 514, Application interpreted construct: Advanced boundary representation;

– Part 515, Application interpreted construct: Constructive solid geometry;

– Part 517, Application interpreted construct: Mechanical design geometric presentation;

– Part 518, Application interpreted construct: Mechanical design shaded representation;

The structure of this International Standard is described in ISO 10303-1. The numbering of the parts of the International Standard reflects its structure:

– Parts 11 to 13 specify the description methods,

- Parts 21 to 26 specify the implementation methods,
- Parts 31 to 35 specify the conformance testing methodology and framework,
- Parts 41 to 49 specify the integrated generic resources,
- Parts 101 to 106 specify the integrated application resources,
- Parts 201 to 232 specify the application protocols,
- Parts 301 to 332 specify the abstract test suites,
- Parts 501 to 518 specify the application interpreted constructs, and

Should further parts of ISO 10303 be published, they will follow the same numbering pattern.

Annexes A, B, C, D, and E forms an integral part of this part of ISO 10303. Annex B is for information only.

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

This International Standard is organized as a series of parts, each published separately. The parts of ISO 10303 fall into one of the following series: description methods, integrated resources, application interpreted constructs, application protocols, application modules, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303-1. This part of ISO 10303 is a member of the description methods series.

This part of ISO 10303 specifies the EXPRESS-X mapping language.

INTERNATIONAL STANDARD © ISO ISO/NWI 10303-1?

Industrial automation systems and integration — Product data representation and exchange — Part 1???: Description methods: The EXPRESS-X language reference manual

1. Scope

This part of ISO 10303 defines a language by which relationships of data defined by models in the EXPRESS language can be specified. The language is called EXPRESS-X.

EXPRESS-X is a structural data mapping language. It consists of language elements that allow an unambiguous specification of the relationship between models.

The following are within the scope of this part of ISO 10303:

- Mapping data defined by one EXPRESS model to data defined by another EXPRESS model.
- Mapping data defined by one version of an EXPRESS model to data defined by another version of EXPRESS model, where the two schemas have different names.
- Specification of requirements for data translators for data sharing and data exchange applications.
- Formal specification of alternate views of data defined by an EXPRESS model.
- Provisions for an alternate notation for application protocol mapping tables.
- Provisions for bi-directional mappings where mathematically possible.
- Concatenation of mappings sharing a common model.
- Specification of constraints evaluated against data produced by mapping.

The following are outside the scope of this part of ISO 10303:

- Mapping of data defined using means other than EXPRESS.
- Mapping of data defined using the second edition of EXPRESS.
- Identification of the version of an EXPRESS schema.

– Graphical representation of constructs in the EXPRESS-X language.

2. Normative references

The following standards contain provisions that, through reference in this text, constitute provisions of this part of ISO 10303. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 10303-1:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles*.

ISO 10303-11:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*.

3. Definitions

3.1 Terms defined in ISO 10303-1

This part of ISO 10303 makes use of the following terms defined in ISO 10303-1.

– data;

– data specification language;

– exchange structure;

– implementation method;

– information;

– information model.

3.2 Terms defined in ISO 10303-11

This part of ISO 10303 makes use of the following terms defined in ISO 10303-11.

– complex entity data type;

- complex entity (data type) instance;
- constant;
- entity;
- entity data type;
- entity (data type) instance;
- instance;
- partial complex entity data type;
- partial complex entity value;
- population;
- simple entity (data type) instance;
- subtype/supertype graph;
- token;
- value.

3.3 Other definitions

For the purposes of this part of ISO 10303, the following definitions apply:

3.3.1 binding: an ordered tuple of values taken from source data entity extents or view extents according to the requirements of a view or map declaration.

3.3.2 binding extent: the set of bindings corresponding to source data entity extents and view extents.

3.3.3 evaluation (of a view or map): the application of a binding to a view or map. Evaluation of a view may produce a view extent. Evaluation of a map may produce entity instances in the target data set.

3.3.4 inverse evaluation (of a view / map): the updating of source data values through the update of (view instance / target entity instance) attribute values. Inverse evaluation shall maintain the relationship between (view instance / entity instance) attribute values and source data as defined in the (VIEW / MAP) declarations.

3.3.5 map: a declaration that defines a relationship between data of one (or more) source entity types and data of one (or more) target entity types.

3.3.6 view: an alternative organization of the information in an EXPRESS model.

3.3.7 view extent: an aggregation data type having as its domain a collection of values of a given view data type. An element of the collection may be identified by a binding.

3.3.8 view data type: a representation of a view. A view data type establishes a domain of values defined by common attributes.

3.3.9 view instance: a named unit of data which represents an alternative organization of source data. It is a member of the domain established by a view entity type.

4. Conformance requirements

4.1 Formal specifications written in EXPRESS-X

4.1.1 Lexical language

A formal specification written in EXPRESS-X shall be consistent with a given level as specified below. A formal specification is consistent with a given level when all checks identified for that level as well as all lower levels are verified for the specification.

Levels of checking

Level 1: Reference checking. This level consists of checking the formal specification to ensure that it is syntactically and referentially valid. A formal specification is syntactically valid if it matches the syntax generated by expanding the primary syntax rule (`syntax`) given in Annex A. A formal specification is referentially valid if all references to EXPRESS-X items are consistent with the scope and visibility rules defined in clauses 10 and 11.

Level 2: Type checking. This level consists of checking the formal specification to ensure that it is consistent with the following:

- expressions shall comply with the rules specified in clause 12 and in ISO 10303-11:1994 clause 12;
- assignments shall comply with the rules specified in ISO 10303-11:1994 clause 13.3.

Level 3: Value checking. This level consists of checking the formal specification to ensure that it is consistent with statements of the form, ‘A shall be greater than B’, as specified in clause 7 to 14. This is limited to those places where both A and B can be evaluated from literals and/or constants.

Level 4: Complete checking. This level consists of checking the formal specification to ensure that it is consistent with all stated requirements as specified in this part of ISO 10303.

4.2 Implementations of EXPRESS-X

4.2.1 EXPRESS-X language parser

An implementation of an EXPRESS-X language parser shall be able to parse any formal specification written in EXPRESS-X, consistent with the conformance class associated with that implementation. An EXPRESS-X language parser shall be said to conform to a particular checking level (as defined in 4.1.1) if it can apply all checks required by that level (and any level below it) to a formal specification written in EXPRESS-X.

The implementor of an EXPRESS-X language parser shall state any constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented for the purpose of conformance testing.

4.2.2 EXPRESS-X mapping engine

An implementation of an EXPRESS-X mapping engine shall be able to evaluate and/or execute any formal specification written in EXPRESS-X, consistent with the conformance class associated with that implementation. The execution and/or evaluation of a mapping is relative to one or more source data sets; the specification of how these data sets are made available to the mapping engine is outside the scope of this part of ISO 10303.

The implementor of an EXPRESS-X mapping engine shall state any constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented for the purpose of conformance testing.

An implementation of an EXPRESS-X mapping engine may take many forms; the following subclauses identify representatively, not exhaustively, the support a mapping engine may provide.

4.2.2.1 Support of VIEW declarations

An implementation shall be said to support VIEW declarations if it meets all of the following criteria:

- The mapping engine accepts a single data set described by an EXPRESS information model.
- The source data instances may be accessed through the evaluation of VIEW declarations.

4.2.2.2 Support of MAP declarations

An implementation shall be said to support MAP declarations if it meets all of the following criteria:

- The mapping engine accepts at least a single data set described by an EXPRESS information model.

– The mapping engine generates a single data set for a given EXPRESS information model.

– The source data instances may be accessed through the evaluation of MAP declarations.

4.2.2.3 Support of the propagation of updates

An implementation shall be said to support the propagation of updates if updates on viewed / mapped data are reflected in source data by the inverse evaluation of the appropriate declarations whenever possible.

Propagation of updates is not possible in situations where any of the following hold:

– The view / target entity is derived from / mapped to two or more source entities by applying a join operation. (For example, the view / target entity `person_in_dept` corresponds to the source entities `person` and `department` where the join condition `person.id = department.person_id` evaluates to true.)

– Duplicates (with respect to value equivalence of attributes) which exist in the source data are eliminated in the view / target data.

– View / target attributes are derived from / mapped to source schema elements by applying mathematical expressions that are not mathematically invertible.

– The view / target schema defines additional subtypes which do not exist in the source schema(s).

– Subtypes which are defined in the souce schema(s) are projected (i.e., not contained) in the view / target schema.

– The sort order of source attributes of type AGGREGATE is eliminated in the view / target schema.

– Duplicates (with respect to value equivalence) of elements of source attributes of type AGGREGATE are eliminated in the view / target schema.

– A single source entity corresponds to a network of interconnected view / target entities (by relationships or equivalence of attribute values[1]).

4.2.2.4 Push mapping

An implementation shall be said to be a push mapping implementation if it meets all of the following criteria:

– The mapping engine accepts one or more source data sets, and produces one or more output data sets.

– The output data sets are derived from the input data sets by the execution and evaluation of potentially all of the VIEW and MAP declarations.

1. The latter kind of relationship is comparable to primary key - foreign key relationships in the relational data model.

– Every instance in the source data sets is mapped as specified in the mapping schema into the output data sets.

4.2.2.5 Pull mapping

An implementation shall be said to be a pull mapping implementation if it meets all of the following criteria:

– The mapping engine accepts one or more source data sets.

– Specified target data instances are derived on demand from the input data sets by the execution and evaluation of the appropriate VIEW or MAP declarations.

NOTE — This part of ISO 10303 does not define how appropriate VIEW / MAP declarations are identified.

4.2.2.6 Support of constraint checking

An implementation shall be said to support constraint checking if it implements the concepts described in clause 9.6 of ISO 10303-11:1994 against entity instances in target populations and against view instances in the view extents.

NOTE — The evaluation of constraints has no effect on the mapping execution model.

4.3 Conformance classes

An implementation shall be said to conform to conformance class 1 if it processes all the declarations that may appear in a SCHEMA_VIEW declaration.

An implementation shall be said to conform to conformance class 2 if it processes all the declarations that may appear in a SCHEMA_MAP declaration that do not contain any external references to a SCHEMA_VIEW declaration.

An implementation shall be said to conform to conformance class 3 if it processes any EXPRESS-X declaration that conforms to this part of ISO 10303.

5. Fundamental principles

The reader of this document is assumed to be familiar with the following concepts, in addition to the concepts described in clause 5 of ISO 10303-11:1994.

EXPRESS-X provides for the specification of:

– alternative views of the data described by an information model described in EXPRESS;

– the transformation of data described by elements of one EXPRESS model into data described by elements of another EXPRESS model.

A SCHEMA_VIEW declaration may provide declarations enabling the specification of the former.

A SCHEMA_MAP declaration may provide declarations enabling the specification of the former and latter.

VIEW and MAP declarations may define a FROM clause. The FROM clause, through its identification of source extents, establishes a binding extent. The binding extent is a set of bindings (ordered tuples) from the cartesian product of source extents. The elements of the tuples are ordered as they appear in the parameter_list of the FROM clause. The values of elements of the binding (elements of a source extent) may be entity instances references, view instance references or values of the primitive EXPRESS types. The binding extent consists only of those tuples for which the application of the WHERE expression of the declaration (an extensional membership predicate) does not returns FALSE.

Bindings and binding extents are notional constructs. There are no means within the language to directly obtain or manipulate bindings or binding extents.

The VIEW declaration may define view attribute declarations. The expression of these declarations are evaluated relative to a given binding.

EXAMPLE 1 — The extents of part and part_usage_approval below are the sets of entity instances (#1,#2,#3) and (#4,#5,#6) respectively. The binding extent of valve_approvers is the set of tuples (<#1,#4>,<#2,#6>). The parameters p and pua are bound to #1 and #4 respectively to produce one view instance of the view extent and #2 and #6 to produce another view instance of that extent.

```
ENTITY part;
  part_number : STRING;
  part_type : STRING;
END_ENTITY;

ENTITY part_usage_approval;
  approver : STRING;
  part_approved : STRING;
END_ENTITY

VIEW valve_approvers;
  FROM (p:part, pua:part_usage_approver)
  WHERE (p.part_number = pua.part_approved) AND
        (p.type = 'valve');

SELECT
  approver : STRING := pua.approver;
  part : STRING := p.part_number;
END_VIEW;
```

```
#1 = PART('p_1','valve');
#2 = PART('p_2','valve');
#3 = PART('p_3','steel door');
#4 = PART_USAGE_APPROVAL('jones','p_1');
#5 = PART_USAGE_APPROVAL('smith','p_3');
#6 = PART_USAGE_APPROVAL('watkins','p_2');
```

In view declarations that do not include an `IDENTIFIED_BY` clause, bindings serves to identify elements of the extent defined by the `VIEW` declaration and source data set.

A schema map written in the EXPRESS-X language describes how elements of one EXPRESS model (the source model) may be transformed into elements of another (the target model). A schema map is composed principally of map and type map declarations. A schema map may reference definitions from an EXPRESS-X schema view.

The specification of a map is based upon an extent of bindings. For each binding in the extent, the body of the map is executed in order to create and populate one or more instances in the target model. A map specification that meets certain criteria is said to be reversible; reversible maps allow a change to data defined by the target model to be propagated back to the source model.

The specification of a type map defines how data described by EXPRESS defined types may be transformed between the source and target model.

EXPRESS function and procedure specifications may form part of an EXPRESS-X specification in order to support the definition of views, maps, or type maps.

The EXPRESS-X language does not describe an implementation environment. In particular, EXPRESS-X does not specify:

– how references to names are resolved;

– how other schemas, schema views, or schema maps are known;

– how input and output data sets are specified;

– how mappings are executed for instances that do not conform to an EXPRESS schema.

6. Language specification syntax

The notation used to present the syntax of the EXPRESS-X language is defined in this clause.

The full syntax for the EXPRESS-X language is given in Annex A. Portions of those syntax rules are reproduced in various clauses to illustrate the syntax of a particular statement. Those portions are not

always complete. It will sometimes be necessary to consult Annex A for the missing rules. The syntax portions within this part of ISO 10303 are presented in a box. Each rule within the syntax box has a unique number toward the left margin for use in cross-references to other syntax rules.

The syntax of EXPRESS-X is defined in a derivative of Wirth Syntax Notation (WSN).

NOTE — See annex B for a reference.

The notational conventions and WSN defined in itself are given below.

```
syntax= { production } .
production= identifier '=' expression '.' .
expression= term { '|' term } .
term= factor { factor } .
factor= identifier | literal | group | option | repetition .
identifier= character { character } .
literal= '''' character { character } '''' .
group= '(' expression ')' .
option= '[' expression ']' .
repetition= '{' expression '}' .
```

– The equal sign `'='` indicates a production. The element on the left is defined to be the combination of the elements on the right. Any spaces appearing between the elements of a production are meaningless unless they appear within a literal. A production is terminated by a period `'.'`.

– The use of an identifier within a factor denotes a nonterminal symbol that appears on the left side of another production. An identifier is composed of letters, digits, and the underscore character. The keywords of the language are represented by productions whose identifier is given in uppercase characters only.

– The word literal is used to denote a terminal symbol that cannot be expanded further. A literal is a sequence of characters enclosed in apostrophes. For an apostrophe to appear in a literal it must be written twice, i.e., `''''`.

– The semantics of the enclosing braces are defined below:

  • curly brackets `'{ }'` indicates zero or more repetitions;

  • square brackets `'[ ]'` indicates optional parameters;

  • parenthesis `'( )'` indicates that the group of productions enclosed by parenthesis shall be used as a single production;

  • vertical bar `'|'` indicates that exactly one of the terms in the expression shall be chosen.

The following notation is used to represent entire character sets and certain special characters which are difficult to display:

– **\a** represents any character from ISO/IEC 10646-1;

– **\n** represents a newline (system dependent) (see clause 7.1.5.2 of ISO 10303-11:1994).

7. Basic language elements

This clause specifies the basic elements from which an EXPRESS-X mapping specification is composed: the character set, remarks, symbols, reserved words, identifiers, and literals.

The basic language elements of EXPRESS-X are those of the EXPRESS language defined in Clause 7 of ISO 10303-11, with the exceptions noted below.

7.1 Reserved words

The reserved words of EXPRESS-X are the keywords and the names of built-in constants, functions, and procedures. Any reserved word in EXPRESS (ISO 10303-11:1994) shall also be a reserved word in EXPRESS-X. The reserved words shall not be used as identifiers. The reserved words of EXPRESS-X are described below.

In the case that a legal EXPRESS identifier is a reserved word in EXPRESS-X, schemas using that identifier can be mapped by renaming the conflicting identifier using the AS keyword in the REFERENCE clause.

7.1.1 Keywords

In addition to the keywords of EXPRESS defined in ISO 10303-11:1994, the following are keywords of EXPRESS-X.

Table 1 — Additional EXPRESS-X keywords

ALIAS	EACH	END_INLINE_FUNCTION	END_MAP
END_SCHEMA_MAP	END_SCHEMA_VIEW	END_TYPE_MAP	END_VIEW
EXTERNAL	IDENTIFIED_BY	IMPORT_MAPPING	INLINE_FUNCTION
MAP	PARTITION	SCHEMA_MAP	SCHEMA_VIEW
SOURCE	TARGET	TYPE_MAP	VIEW

8. Data types

This clause defines the data types provided as part of the language. Every view attribute, map attribute, or type map has an associated data type.

The data types are the same as those for the EXPRESS language defined in clause 8 of ISO 10303-11:1994, with the exceptions noted below.

8.1 Complex entity data type

Complex entity data types are established implicitly by entity declarations related in an inheritance hierarchy (see ISO 10303-11:1994, clause 9.2). An entity data type is assigned an entity identifier by the user. An entity data type is referenced by this identifier. A complex entity data type is referenced by an expression that lists the partial complex entity data types that are combined to form it, separated by the keyword AND.

The partial complex entity data types may be listed in any order.

Any partial complex entity data types that are included in another partial complex entity data type via inheritance are not listed.

Syntax:

```
42 complex_entity_spec = entity_reference AND entity_reference { AND
   entity_reference }.
```

Rules and restrictions:

a) Each entity_ref shall be a reference to an entity which is visible in the current scope.

b) The referenced complex entity data type shall describe a valid domain within some schema (see ISO 10303-11:1994, annex B).

c) A given entity_ref shall occur at most once within a complex_entity_ref.

EXAMPLE 2 — Given the following entity declarations:

```
ENTITY super SUPERTYPE OF ONEOF(a,c);
END_ENTITY;

ENTITY a SUBTYPE OF (super);
END_ENTITY;

ENTITY b SUBTYPE OF (super);
END_ENTITY;

ENTITY c SUBTYPE OF (super);
END_ENTITY;
```

The following are valid complex entity data types:

```
a AND b
b AND c
```

The following are not valid complex entity data types:

```
a AND b AND super
a AND c
b AND c AND c
```

8.2 View data type

View data types are established by `VIEW` declarations (see clause 9.4). A view data type is assigned an identifier in the defining schema map or schema view. A view data type is referenced by this identifier.

A value of a view data type is a view instance and may be produced by evaluating an explicit binding expression.

NOTE — A single view_reference identifier designates both a view data type and an extent data type. The intended construct in any particular situation may be discerned through examination of the EXPRESS-X grammar.

Syntax:

```
138 view_reference = view_ref | primary_extended '.' view_qualifier .
```

8.3 Extent data type

Extent data types are established explicitly by `VIEW` declarations (see clause 9.3) and implicitly by source EXPRESS schema `ENTITY` declarations (ISO 10303-11;1994, clause 8.3.1). An extent data type is assigned an identifier in the defining schema map or schema view. A extent data type is referenced by this identifier.

NOTE — A single view_reference identifier designates both a view data type and an extent data type. The intended construct in any particular situation may be discerned through examination of the EXPRESS-X grammar.

Syntax:

```
 48 extent_reference = source_entity_reference | view_reference .
```

Rules and restrictions:

a) `extent_reference` shall be a reference to an extent which is visible in the current scope.

EXAMPLE 3 — The following declaration defines a view data type and extent data type, each designated by `circle`.

```
VIEW circle;
  FROM (e : ellipse);
  WHERE (e.major_axis = e.minor_axis);
END_VIEW;
```

9. Declarations

This clause defines the various declarations available in EXPRESS-X. An EXPRESS-X declaration creates a new EXPRESS or EXPRESS-X item and associates an identifier with it. The item may be referenced elsewhere by writing the name associated with it.

The principle capabilities of EXPRESS-X are found in the following declarations:

– View;

– Map;

– Schema_view;

– Schema_map;

– Type_map.

In addition, an EXPRESS-X specification may contain the following declarations defined in ISO 10303-11:1994:

– Constant;

– Function;

– Procedure;

– Rule.

Mapping declarations are always explicit.

9.1 Schema_view declaration

A schema_view declaration defines a common scope for a collection of related mapping declarations. A schema_view may contain the following kinds of declarations:

– constant declaration (clause 9.5);

– function declaration (clause 9.6);

– procedure declaration (clause 9.7);

– rule declarations (clause 9.10);

– view declaration (clause 9.3).

The order in which declarations appear within a schema_view declaration is not significant.

Declarations in one schema_view or EXPRESS schema may be made visible within the scope of another schema_view via an interface specification as described in clause 11.

Syntax:

```
100  schema_view_decl = SCHEMA_VIEW schema_view_id { reference_clause } [
     constant_decl ] schema_view_body_element_list END_SCHEMA_VIEW ';' .
 87  reference_clause = REFERENCE FROM foreign_ref [ '(' resource_or_rename
     { ',' resource_or_rename } ')' ] ';' .
 98  schema_view_body_element = function_decl | procedure_decl | view_decl
     | create_view_decl .
```

EXAMPLE 4 — `ap203_arm` names a `schema_view` that may contain declarations defining a view over the schema `config_control_design` in terms of the domain expert's understanding of the information requirements.

```
SCHEMA_VIEW ap203_arm;
REFERENCE FROM config_control_design;
VIEW part_version ...
(* other mapping declarations as appropriate *)
END_SCHEMA_VIEW;
```

9.2 Schema_map declaration

A `schema_map` declaration defines a common scope for a collection of related mapping declarations.

EXAMPLE 5 — `iges2step` names a `schema_map` that may contain declarations for translating geometry defined using and EXPRESS model base upon IGES into a model based on ISO 10303-203.

```
SCHEMA_MAP iges2step;
REFERENCE FROM iges_express_schema;
MAP iges_structure ...
(* other mapping declarations as appropriate *)
END_SCHEMA_MAP;
```

The order in which declarations appear within a schema_map declaration is not significant. In particular, the order of the declarations has no effect upon the resulting mapping.

Declarations in one schema_map may be made visible within the scope of another schema_map via an interface specification as described in clause 11.

A schema_map may contain the following kinds of declarations:

– constant declaration (clause 9.5);

– function declaration (clause 9.6);

– procedure declaration (clause 9.7);

– type_map declaration (clause 9.8);

– view declaration (clause 9.3);

– map declaration (clause 9.4);

– rule declaration (clause 9.10).

Syntax:

```
 92  schema_map_decl = SCHEMA_MAP schema_map_id target_interface_spec
     source_interface_spec { map_interface_spec } { external_functions_spec
     } { type_mapping_stmt } [ constant_decl ] schema_map_body_element_list
     END_SCHEMA_MAP ';' .
113  target_interface_spec = TARGET schema_ref_or_rename { ','
     schema_ref_or_rename } ';' .
105  source_interface_spec = SOURCE schema_ref_or_rename { ','
     schema_ref_or_rename } ';' .
 77  map_interface_spec = IMPORT_MAPPING schema_map_or_view_ref_or_rename {
     ',' schema_map_or_view_ref_or_rename } ';'.
 49  external_functions_spec = EXTERNAL function_head { function_head }
     END_EXTERNAL ';' .
117  type_mapping_stmt = TYPE_MAP target_type_reference {
     target_type_reference } FROM source_type_reference {
     source_type_reference } ';' { type_map_stmt_body } END_TYPE_MAP ';' .
 90  schema_map_body_element = function_decl | procedure_decl | view_decl |
     create_view_decl | map_decl | create_map_decl .
```

The body of a schema_map shall have the same form as the body of a schema in ISO 10303-11;1994, with the following exceptions:

– The schema_map shall include at least one MAP declaration.

– The schema_map shall include a target_interface_spec declaration.

– The schema_map shall include a source_interface_spec declaration.

– The schema_map shall not include the interface_specification declaration (defined in ISO 10303-11;1994).

– The schema_map shall not include the entity declaration (defined in ISO 10303-11;1994).

– The schema_map shall not include the type declaration (defined in ISO 10303-11;1994).

EXAMPLE 6 — This example illustrates the use of required EXPRESS-X declarations. t1, t2, t3, s1 and s2 designate EXPRESS schema. other_map designates an EXPRESS-X schema.

```
SCHEMA_MAP map_name;
  TARGET t1, t2, t2;
  SOURCE s1, s2;
IMPORT MAPPING other_map;
END_SCHEMA_MAP;
```

9.3 Common clauses of the VIEW and MAP declarations

The VIEW and the MAP declarations have the following clauses in common.

9.3.1 The FROM clause

The FROM clause identifies source extents (view extents and entity extents) from which a binding extent is computed.

Syntax:

```
55 from_clause = FROM ( '(' from_parameter_list ')' | from_parameter_list
   ) .
57 from_parameter_list = from_parameter { ',' from_parameter } .
56 from_parameter = [ parameter_id { parameter_id } ':' ] extent_reference
   .
```

Rules and restrictions:

a) parameter_ids shall be unique within the scope of the MAP or VIEW declaration.

9.3.2 The WHERE clause

The WHERE clause defines an extensional membership predicate (an expression) in terms of parameters bound by the FROM clause. The WHERE clause, together with the source extents identified in the FROM clause, defines the binding extent. A tuple is a member of the binding extent unless one or more domain rule expressions of the WHERE clause returns FALSE for the binding of parameters representing that tuple.

The syntax of the WHERE clause is as defined in ISO 10303-11;1994, clause 9.2.2.2.

9.3.3 Identification of view and target instances

The IDENTIFIED_BY clause and the binding defined by the FROM clause provide mutually exclusive constructs to identify view instances (in its usage in the VIEW) and target instances (in its usage in the MAP). These constructs are used to uniquely identify view instances and target instances in explicit binding calls (clause 12.1).

The IDENTIFIED_BY clause declares the structure of an internal ID. The internal ID is a key produced by the concatenation of values resulting from the evaluation of the expressions of the identified_by_clause. The internal ID is a notional construct. There are no means within the language to directly obtain or manipulate an internal ID.

Syntax:

```
58  identified_by_clause = IDENTIFIED_BY expression { ',' expression } ';'.
```

Rules and restrictions:

a) expression shall not evaluate to a value of type AGGREGATE.

The internal ID provides the following functionality:

- View instances or generated target instances can be related by the internal ID to the corresponding source instances because the internal ID is built by an expression over source attributes values / OIDs.
- The mapping engine automatically avoids unintended duplicates in the view / target. That is, independent of the execution order of the mapping and independent of the starting point, it can be guaranteed that the same view / target instance is not generated more than once.
- Support of uniqueness constraints for view / target instances based on source information (and therefore implicitly also on view / target information because it is derived / mapped from the sources).
- The internal ID is used to specify relationship between view / target instances by the means of explicit binding (clause 12.1).

EXAMPLE 7 — In the following, the source data set may contain multiple value equivalent instances of `number` whereas the target population shall contain no value equivalent instances of `unique_number`. Were no `IDENTIFIED_BY` clause specified for this example, the OID of the source entity `number` (the binding) would have served as the internal ID and thus the target instances of `number` may not have been value unique.

```
source schema:                target schema:
ENTITY number;                ENTITY unique_number;
  its_value : INTEGER;          its_value : INTEGER;
END_ENTITY;                   END_ENTITY;
```

```
MAP unique_number
FROM n : number
IDENTIFIED_BY n.its_value;
END_MAP;
```

NOTE — The declaration of a unique_number VIEW is similar to the MAP unique_number in EXAMPLE 7.

Although the internal ID uniquely identifies view instances and target instances, an internal ID may not necessarily uniquely identify a binding in the binding extent. When a one-to-many relationship exists between internal IDs and bindings, the following procedure is used to produce the view instance or target instances corresponding to the unique internal ID:

If the `IDENTIFIED_BY` clause and sources extents are such that two or more bindings correspond to a single internal ID, then the values of attributes of the resulting view instance or entity instances is computed as follows:

– If for each such binding, the evaluation of the `view_attr_assgnmt_expr` (or `map_attr_assgnmt_expr` in the case of a `MAP`) of the attribute produces an equal value, that value is assigned to the attribute.

– If for two or more bindings, the evaluation of the `view_attr_assgnmt_expr` (or `map_attr_assgnmt_expr` in the case of a `MAP`) of the attribute produces unequal values, an indeterminate value is assigned to the attribute.

EXAMPLE 8 — Assuming that one instance of the target entity `department` corresponds to a set of instance of `employee` where all of them have the same value for the attribute `dept`.

```
source schema:                target schema:
ENTITY employee;              ENTITY department;
  name : STRING;                employees : SET OF STRING;
  manager : STRING;             manager : STRING;
  dept : STRING;                name : STRING;
END_ENTITY;                   END_ENTITY;
```

```
mapping:
MAP department
FROM e : employee
IDENTIFIED_BY e.dept;
SELECT
  name := e.dept;
  manager := e.manager;
  employees := e.name;
END_MAP;
```

Assuming that each department has exactly one manager and more than one employees, each instance of the target entity department will have a value for the attributes name and manager but an indeterminate value for employees.

NOTE — It will be shown below how the inline view resp. the for expression can be used to assign values to a view / target attribute of type AGGREGATE and therefore to fold all source instances of employee which have the same value for dept into one target instance of department.

9.4 View declaration

A `VIEW` declaration creates a view data type and declares an identifier to refer to it. A `VIEW` declaration defines one or more view expressions that may be evaluated for a given binding in the binding extent.

A `VIEW` declaration consists of one or more partitions. Each partition defines part of the entire view extent. Partitions may be named; the name is optional if there is only a single partition.

EXAMPLE 9 — The following view collects the information about persons serving in roles within organizations. This information is collected from two instances of person_and_organization and cc_design_person_and_organization_assignment. The two instances must be related via the assigned_person_and_organization attribute of the cc_design_person_and_organization_assignment. Three attribute reference expressions are given.

```
VIEW arm_person_role_in_organization;
FROM (pao : person_and_organization,
      ccdpaoa : cc_design_person_and_organization_assignment)
WHERE ccdpaoa.assigned_person_and_organization :=: pao;
SELECT
  person : person := pao.the_person;
  org : organization := pao.the_organizaion;
  role : label := ccdpaoa.role.name;
END_VIEW;
```

Syntax:

```
130  view_decl = VIEW view_id ';' [ subtype_of_clause ] ( view_partitions |
     view_decl_body ) END_VIEW ';' .
134  view_partition = PARTITION [ partition_id ':' ] view_decl_body .
131  view_decl_body = [ from_clause ] [ identified_by_clause ] [
     where_clause ] [ view_project_clause ] .
```

Rules and restrictions:

a) If in a view_decl a subtype_of_clause is specified, no from_clause shall be declared in the `view_decl_body`s of any partition.

b) If no subtype_of_clause is specified, the from_clause in any view_decl_body of this view_decl is mandatory.

c) Each attribute expression reference declared in the view declaration shall be unique within the declaration.

9.4.1 View attributes

An attribute of a view data type represents a property whose value is computed as the evaluation of its `view_attr_assgnmt_expr`, an expression, in order to create a view instance. Each view attribute declaration identifies a distinct property.

The name of a view attribute (`view_attribute_id`) represents the role played by it associated value in the context of the view in which it appears.

The expression represented by a `view_attr_assgnmt_expr` is evaluated in the context of a given binding in the binding extent. The evaluation may produce a reference to source data, a full or partial view extent, or an aggregate of type SET with a base type that is an entity data type or view data type.

Syntax:

```
136  view_project_clause = SELECT (extent_reference |
     view_attr_decl_stmt_list) .
120  view_attr_decl_stmt_list = view_attribute_decl { view_attribute_decl }
     .
121  view_attribute_decl = view_attribute_id ':' [ source_schema_ref '.' ]
     base_type ':=' view_attr_assgnmt_expr ';' .
119  view_attr_assgnmt_expr = expression | view_cond_attr_expr |
     view_case_expr | inline_view_decl | view_call .
```

Rules and restrictions:

a) The expression (`expression`, `view_cond_attr_expr`, `view_case_expr`, `inline_view_decl`, `view_call`) shall be assignment compatible with the data type of

the view attribute.

b) Every view attribute of a view instance shall have a value.

EXAMPLE 10 — `circle` names a view extent defined to contain all ellipse instances with equal length major and minor axes. For a given binding in the binding extent, the mapping engine evaluates the expressions `e.semi_axis_1` and `e.position` to obtain values of the `radius` and `position` view attributes, respectively.

```
VIEW circle;
FROM (e : ellipse) WHERE e.semi_axis_1 = e.semi_axis_2;
SELECT
  radius : positive_length_measure := e.semi_axis_1;
  position : axis2_placement := e.position;
END_VIEW;
```

9.4.2 View partitions

A view extent is the union of the extents defined by its partitions. If the `VIEW` declaration contains more than one partition, the partitions shall be named. A `partition_id` names a partition.

EXAMPLE 11 — In ISO 10303-201, the application object `organization` may be mapped to either a `person`, an `organization`, or both a `person` and `organization` entity in the AIM. This is specified in EXPRESS-X as follows:

```
VIEW arm_organization
PARTITION a_single_person :
  FROM (p : person)
    ...
PARTITION a_single_organization :
  FROM (o : organization)
    ...
PARTITION a_person_in_an_organization :
  FROM (po : person_and_organization)
END_VIEW;
```

Syntax:

```
134 view_partition = PARTITION [ partition_id ':' ] view_decl_body .
```

Rules and restrictions:

a) All partitions of a `VIEW` declaration shall define the same attributes (including names and types)

b) The attributes of a VIEW declaration shall appear in the same order in each of its partitions..

9.4.3 Specifying subtype views

EXPRESS-X allows for the specification of views as subtypes of other views. A view is a subtype if it contains a SUBTYPE declaration. The extent of a subtype view is a subset of the extent of its supertype as defined by the extensional membership predicate defined by the WHERE clause in the subtype.

A VIEW declaration containing a SUBTYPE declaration shall not contain a FROM declaration.

A subtype VIEW may inherit attributes from its supertype. Inheritance of attributes shall adhere to the rules and restrictions of attribute inheritance defined in ISO 10303-11;1994 clause 9.2.3.3.

A subtype VIEW declaration may redefine attributes found in one of its supertypes. The redefinition of attributes shall adhere to the rules and restrictions of attribute redistribution defined in ISO 10303-11;1994 clause 9.2.3.4.

EXAMPLE 12 — The following view illustrates subtyping. The view male defines an additional membership requirement (gender = 'M') for view instances of the subtype.

```
VIEW person;
FROM ...
END_VIEW;

VIEW male SUBTYPE OF person;
WHERE gender = 'M';
...
END_VIEW;
```

9.5 Map declaration

The MAP declaration supports the specification of correspondences between semantically equivalent elements of two or more EXPRESS models possessing differing structure. Each MAP declaration specifies how base instances of one or more types are to be mapped to target instances. That is, a MAP declaration supports, in a single declaration, the mapping from many target entities to many source entities.

A MAP declaration consists of a header and body of one or more map statements. The purpose of the header is to define the conditions under which one or more new target instance(s) should be created from one or more instances in a base model. The map_decl_body defines how the values of the attributes for a newly created instance are to be computed. The concept of alternative views, i.e. partitions, is also available for maps.

Syntax:

```
 74  map_decl = MAP map_decl_header  ( (map_decl_body { map_partitions }) |
     map_decl_body )  END_MAP ';' .
 78  map_partition = PARTITION [ partition_id ':' ] map_decl_body .
 75  map_decl_body = ( ( from_clause [ identified_by_clause ] ) |
     subtype_of_clause ) [ where_clause ] { entity_instantiation_loop } [
     map_project_clause ] .
```

9.5.1 Header of the MAP declaration

The header identifies one or more entity types defined explicitly or implicitly in the target EXPRESS schema. It is not required that those target entity types are related to each other by relationships.

Syntax:

```
 76  map_decl_header = target_entity_ref_list [ NAMED network_id ] .
111  target_entity_ref_list_el = [ target_entity_alias_id ':' [ LIST
     bound_spec OF ] ] target_entity_reference .
112  target_entity_reference = entity_reference | complex_entity_spec |
     target_schema_ref '.' '(' complex_entity_spec ')' .
```

Rules and restrictions:

a) For each entity type appearing in the `target_entity_ref_list` none of its supertypes shall appear in the list.

A target entity type shall not be mapped in more than one `MAP` declaration in which the headers of those declarations consist only of a single target entity type. However, one target entity can be mapped in more than one `MAP` declarations (say n), if n-1 `MAP` declarations are group mappings. To support the call of a target entity mapping inside a group mapping, the MAP declaration of the group mapping is given a name (`group_name`).

NOTE — A single target entity type may be mapped in various ways by means of partitions.

EXAMPLE 13 — In the example below, a pump in the source data model is mapped to a set of target entities.

```
MAP xx AS group_for_pump
FROM p : pump
  -- attribute mappings of the target entities
END_MAP;
```

The initial values of the attributes of the newly created instance(s) are indeterminate.

9.5.2 The SELECT clause

The SELECT clause is needed if source attributes have to be projected in the view / target entity or if a specific entity of multiple FROM clause entities have to be mapped to the view / target entity identically (same attributes with same values). In the latter case, just this entity is specified after the SELECT keyword. If view / target attributes are not identical to the source schema due to their structure or due to their values, then the SELECT clause contains the attribute assignment statements to specify which view / target attributes have to be built by which expression over the source data. Those attribute assignment statements are different for the VIEW and the MAP declaration as shown in the subsequent sections.

The MAP declaration SELECT clause identifies data that shall appear in the target data set.

The syntactic form SELECT extent_reference declares that an entity instance value equivalent to that bound to extent_reference shall appear in the target data set.

The syntactic form SELECT map_attribute_decl_block assigns values (map_attr_assgnmt_expr) to the target entity attributes (l-values) identified by the syntactic form [entity_reference] '.' attribute_ref .

Syntax:

```
80 map_project_clause = SELECT ( extent_reference  |
   map_attribute_decl_block ).
68 map_attribute_decl_block = map_attr_decl_stmt_list .
67 map_attr_decl_stmt_list = map_attribute_declaration {
   map_attribute_declaration } .
69 map_attribute_declaration = [ entity_reference '.' ] attribute_ref
   ':=' map_attr_assgnmt_expr ';' .
```

9.5.3 Partitions within a MAP declaration

The partition concept is the same as described within the VIEW declaration (see clause 9.3.1). It is extended so that partitions can be defined for a list of target entities.

If multiple target entities are listed in the header of the MAP declaration, different subset of those entities can be used for the partitions.

Syntax:

```
78 map_partition = PARTITION [ partition_id ':' ] map_decl_body .
```

Rules and restrictions:

a) If the MAP declaration contains more than one partition, the partitions shall be named.

b) All partitions must define the same attributes (`attribute_ref`) and types.

9.5.4 Inheritance

If an inheritance hierarchy is defined in the target EXPRESS schema, the MAP declaration of the supertype must specify the mapping for all instances of this supertype, i.e. all direct instances and all instances of all its subtypes. If the mapping for the subtypes is not the same, partitions must be specified.

EXAMPLE 14 — Inheritance for MAP declaration.

```
target schema:            source schema:
ENTITY person;            ENTITY male;
  name : STRING;            name : STRING;
END_ENTITY;               END_ENTITY;

ENTITY male;              ENTITY female;
SUBTYPE_OF person;          name : STRING;
END_ENTITY;               END_ENTITY;

ENTITY female;
SUBTYPE OF person;
END_ENTITY;

mapping specification:
MAP person
PARTITION female_person :
  FROM female
PARTITION male_person :
  FROM male
END_MAP;

MAP male
SUBTYPE OF person
PARTITION male_person
END_MAP;

MAP female;
SUBTYPE OF person
PARTITION female_person
END_MAP;
```

Alternatively, the clauses inside the partition can be specified in the subtypes and only referenced in the supertype. Once they are defined, the constraints which are specified for an specific view can be extended in subviews.

The mapping of supertypes or subtypes cannot be specified in a network mapping.

9.6 Create declaration

The CREATE declaration defines the form of an entity that shall be created in the target data set.

Syntax:

```
43 create_map_decl = CREATE instance_id INSTANCE_OF
    target_entity_reference ';' map_attr_decl_stmt_list END_CREATE ';' .
```

9.7 Constant declaration

Constants may be defined for use within the WHERE clause of a view or map declaration, or within the body of a map declaration or algorithm.

Constant declarations are as defined in ISO 10303-11:1994 clause 9.4.

9.8 Function declaration

Functions may be defined for use within the WHERE clause of a view or map declaration, or within the body of a map declaration.

Function declarations are as defined in ISO 10303-11:1994 clause 9.5.1.

9.9 Procedure declaration

Procedures may be defined for use within the body of a map declaration.

Procedure declarations are as defined in ISO 10303-11:1994 clause 9.5.2.

9.10 Rule declaration

Rules may be defined for use within the SCHEMA_VIEW and SCHEMA_MAP clause.

Rule declarations are as defined in ISO 10303-11:1994 clause 9.6.

9.11 Type map declaration

A type map declaration specifies how a value of a defined type is mapped to a value of another type within the scope of a schema map.

Syntax:

```
117  type_mapping_stmt = TYPE_MAP target_type_reference {
     target_type_reference } FROM source_type_reference {
     source_type_reference } ';' { type_map_stmt_body } END_TYPE_MAP ';' .
116  type_map_stmt_body = [ schema_ref '.' ] base_type ':='
     type_assgnmt_expr ';'.
```

EXAMPLE 15 — The following specifies the mapping between the types `dollar` and `dmark`.

```
TYPE_MAP dmark FROM dollar;
  dmark := 1.5 * dollar;
  dollar := dmark / 1.5;
END_TYPE_MAP;
```

The mapping is applied whenever the source attribute type is type compatible with one of the first types and the target attribute type is type compatible with one of the second types.

Rules and restrictions:

a) Body is not needed if just renaming.

b) No more than two expressions; if second is omitted then reverse mapping is implicit.

c) The two expressions shall be inverses of each other.

d) No entity instances shall be mapped by the TYPE_MAP. The base type shall not be an entity type.

10. Scope and visibility

An EXPRESS-X declaration creates an identifier that can be used to reference the declared item in other parts of the schema_view (or in other schema_views). Some EXPRESS-X constructs implicitly declare items, attaching identifiers to them. In those areas where an identifier for a declared item may be referenced, the declared item is said to be visible. An item may only be referenced where its identifier is visible. For the rules of visibility, see clause 10.2 For further information on referring to items using their identifiers, see clause 12.

Certain EXPRESS-X items define a region (block) of text called the scope of the item. This scope limits the visibility of identifiers declared within it. Scope can be nested; that is, an EXPRESS-X item which establishes a scope may be included within the scope of another item. There are constraints on which items may appear within a particular EXPRESS-X item’s scope. The constraints are usually enforced by the syntax of EXPRESS-X.

For each of the items specified in table 2 below the following subclauses specify the limits of the scope defined, if any, and the visibility of the declared identifier both in general terms and with specific details.

Table 2 — Scope and identifier defining items

Item	Scope	Identifier
view attribute		•
view	•	•
partition	•	•
schema_view	•	•

10.1 Scope rules

The general scope rules are as defined in ISO 10303-11:1994.

10.2 Visibility rules

The general visibility rules are as defined in ISO 10303-11:1994.

10.3 Explicit item rules

The following clauses provide more detail on how the general scoping and visibility rules apply to the various EXPRESS-X items.

10.3.1 Schema_view

Visibility: A schema_view identifier is visible to all other schema_views.

Scope: A schema_view declaration defines a new scope. This scope extends from the keyword SCHEMA_VIEW to the keyword END_SCHEMA_VIEW that terminates that schema_view declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a schema_view declaration:

– constant;

– function;

– map;

– procedure;

– rule;

– type_map;

– view.

10.3.2 View

Visibility: A view identifier is visible in the scope of the function, procedure, rule, or schema_view in which it is declared. A view identifier remains visible within inner scopes which redeclare that identifier.

Scope: A view declaration defines a new scope. This scope extends from the keyword VIEW to the keyword END_VIEW which terminates that entity declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a view declaration:

– view expression;

– partition label.

10.3.3 View partition label

Visibility: A partition label is visible in the scope of the view in which it is declared.

10.3.4 View expression

Visibility: A view expression identifier is visible in the scope of the view in which it is declared.

11. Interface specification

This clause specifies the constructs that enable items declared in one schema, schema_view, or schema_map to be visible in another schema_view or schema_map. The REFERENCE specification allows enables item visibility.

Syntax:

```
 77  map_interface_spec = IMPORT_MAPPING schema_map_or_view_ref_or_rename {
     ',' schema_map_or_view_ref_or_rename } ';'.
 94  schema_map_or_view_ref_or_rename = schema_map_ref_or_rename |
     schema_view_ref_or_rename .
 95  schema_map_ref_or_rename = [ schema_map_alias_id ':' ] schema_map_ref
     .
102  schema_view_ref_or_rename = [ schema_view_alias_id ':' ]
     schema_view_ref .
```

A foreign declaration is any declaration which appears in a foreign schema, schema_view, or schema_map (which is not the current schema_view or schema_map).

A foreign EXPRESS or EXPRESS-X item may be given a new name in the current schema_view or schema_map. The item shall be referred to in the current schema by the new name if given following the AS keyword. This can be used in order to rename EXPRESS items that would otherwise use EXPRESS-X reserved words as their identifier.

11.1 Reference interface specification

A REFERENCE specification enables the following items, declared in a foreign schema, schema_view, or schema_map, to be visible in the current schema_view or schema_map:

- View;
- Map;
- Type_map;
- Constant;
- Entity;
- Function;
- Procedure;
- Type.

The REFERENCE specification gives the name of the foreign schema, and optionally the names of EXPRESS or EXPRESS-X items declared therein. If there are no names specified, all the items declared in the foreign schema, schema_view, or schema_map are visible within the current schema_view or schema_map.

Syntax:

```
 87  reference_clause = REFERENCE FROM foreign_ref [ '(' resource_or_rename
     { ',' resource_or_rename } ')' ] ';' .
 53  foreign_ref = schema_ref | schema_view_ref | schema_map_ref .
```

Rules and restrictions:

11.2 Implicit interfaces

11.3 SCHEMA_MAP interfaces

A schema_map interface specification allows items defined in foreign schema to be visible within the schema map. It also define the source and target schemas.

Syntax:

```
 92  schema_map_decl = SCHEMA_MAP schema_map_id target_interface_spec
     source_interface_spec { map_interface_spec } { external_functions_spec
     } { type_mapping_stmt } [ constant_decl ] schema_map_body_element_list
     END_SCHEMA_MAP ';' .
```

11.3.1 Source schema interface

The source schema interface specifies the name of the source schema.

Syntax:

```
105  source_interface_spec = SOURCE schema_ref_or_rename { ','
     schema_ref_or_rename } ';' .
```

11.3.2 Target schema interface

The target schema interface specifies the name of the target schema.

Syntax:

```
113  target_interface_spec = TARGET schema_ref_or_rename { ','
     schema_ref_or_rename } ';' .
```

11.3.3 Map interface

The map interface specifies how one SCHEMA_MAP may reference another.

Syntax:

```
77 map_interface_spec = IMPORT_MAPPING schema_map_or_view_ref_or_rename {
   ',' schema_map_or_view_ref_or_rename } ';'.
```

11.3.4 External functions

The external functions interface specifies the interface to a function that is not defined in EXPRESS but is defined externally and will be called as part of the mapping.

EXAMPLE 16 — A mapping may call an external function to convert geometry from advanced BREP into a faceted representation.

Syntax:

```
49 external_functions_spec = EXTERNAL function_head { function_head }
   END_EXTERNAL ';' .
```

12. Expressions

Expressions are combinations of operators, operands, and function calls that are evaluated to produce a value. Anything that is an expression as defined in ISO 10303-11:1994 clause 12 is also a valid expression in EXPRESS-X. In addition, the following subclauses describe how expressions may be used with bindings and extents.

Precedence of operators and the order of evaluation of expressions are as defined in ISO 10303-11:1994 clause 12.

Entity constructors create instances that are local only to the function or procedure and do not exist in either the target or the source.

12.1 Explicit binding

The main intention behind the explicit binding is to specify relationships between view / target instances within the mapping specification. That is, this concept supports the generation of links between view / target instances based on the assumption that the corresponding source instances are also related (via some path expressions or equivalence of attribute values). The explicit binding expres-

sion is specified for the attribute assignment statement of the attribute which represents the relationship in the view / target entity. The generation of such a relationships requires the name of the view / target entity which is referenced and some expression to select the specific instance(s) to be referenced. That is, it has to be specified on the schema level which views / target entities have to be related and it has to be specified also on the instance level which instances of those views / target entities have to be related.

This relationship is specified in a function-like style where the name of the referenced view / target entity is the function name (the so-called explicit-binding operator's name).

The so-called expression of the explicit-binding operator is then used to select the specific instances which have to be referenced. For this purpose, the expression is evaluated and the resulting value is them compared with the value of the internal ID(s) of the referenced entity (specified by the IDENTIFIED_BY clause). An example is given below.

EXAMPLE 17 — Explicit binding concept.

```
source schema:
ENTITY approval;
  id : STRING;
  creator : STRING;
END_ENTITY;
```

```
source instance set:
#1 = approval('a_1','miller');
#2 = approval('a_2','jones');
#3 = approval('a_3','miller');
```

```
target schema:
ENTITY person;
  id : STRING;
END_ENTITY;
```

```
ENTITY design_order;
  id : STRING;
  approved_by : person;
END_ENTITY;
```

In this mapping example, it is assumed that `design_order.id` corresponds to `approval.id`, `person.id` corresponds to `approval.creator`, and `design_order` references `person` via the attribute `approved_by` where `design_order` and `person` correspond to the same `approval`. Furthermore, it is assumed that if more than one source instance of `approval.creator` exist with the same, only one target instance of `person` has to be generated. This is achieved by the following VIEW declaration.

```
MAP person
FROM a : approval
IDENTIFIED_BY a.creator
SELECT
  id := a.creator;
  of_design_order := design_order(a);
END_MAP;
```

The IDENTIFIED_BY clause ensures that target instances of `person` are unique w.r.t. the source attribute `approval.creator`. That is, from source instances of `approval` having the same value of `creator` only one target instance of `person` is generated. The target instanes of `person` are internally identified by the attribute value `approval.creator` of the corresponding source instances.

```
MAP  design_order
FROM a : approval
IDENTIFIED_BY OID(a) -- optional because default
SELECT
  id := a.id;
  approved_by := person(a.creator); -- explicit binding
END_MAP;
```

The name of the explicit-binding operator states that target instances of `design_order` have to be linked to `person` via the attribute `approved_by`. So far, it is not clear which target instances of `design_order` have to be linked to which instances of `person`. This is done by the expression of the explicit-binding operator. The value of `approval.creator` is compared with the internal IDs of the target instances of `person`. For example, the target instance `#3` of `design_order` has to be linked to the target instance `#1` of `person`, because this `design_order` is mapped from the source instance of `approval` having the value `'miller'` for `approval.creator` and this is the internal ID of target instance `#1`.

```
Generated target instance set:
#1 = person('miller'); -- internal ID 'miller', mapped from #1,#3
#2 = person('jones');  -- internal ID 'jones', mapped from #2
#3 = design_order('a_1',#1); -- internal ID #1, mapped from #1
#4 = design_order('a_3',#2); -- internal ID #2, mapped from #2
#5 = design_order('a_3',#1); -- internal ID #3, mapped from #3
```

NOTE — The same concept is supported by the MAP declaration.

The explicit binding has to be specified using the following syntax.

Syntax:

```
70 map_call = entity_reference [ '@' network_or_partition_qualification ]
   '(' expression { ',' expression } ')' .
82 network_or_partition_qualification = network_ref | [ network_ref '.']
   partition_ref .
```

An explicit binding is used to specify a particular member of an extent. by providing instances which are bound to the variables in the FROM clause, using a function-like syntax. The result of an explicit binding is a binding data type.

If an IDENTIFIED_BY clause is present in the definition of the extent, then the arguments of the explicit binding shall match that clause.

EXAMPLE 18 — Explicit bindings are useful for describing a relationship between two views: :

```
VIEW my_line;
FROM (l : line);
SELECT
  point : my_point := my_point(l.pnt);
  ...
END_VIEW;
VIEW my_point;
FROM (cp : cartesian_point)
  ...
```

12.2 Partial explicit binding

A partial explicit binding is an explicit binding in which one or more of the parameters is indeterminate. The result of a partial explicit binding is the subset of the extent that matches the parameter values that are provided.

EXAMPLE 19 — In the following, the various versions associated with a part are collected by using a partial explicit binding. The result will be the subset of the extent for which the second component of the binding is equal to the specified product instance.

```
VIEW part;
FROM (p : product)
SELECT
  versions : SET OF version_and_its_product
     := version_and_its_product(?, p);
END_VIEW;
VIEW version_and_its_product;
FROM (pdf : product_definition_formation, p : product)
WHERE p :=: pdf.of_product;
SELECT
  the_version : product_definition_formation := pdf;
END_VIEW;
```

12.3 Inline views

An inline view is the definition and evaluation of a view simultaneously. Inline views do not have names, and may not be bound to explicitly. The result of an inline view is an extent (i.e., a set of bindings).

EXAMPLE 20 — In the following example the versions of a part are collected by using an inline view.

```
VIEW part
FROM (p : part)
SELECT
  versions : SET OF product_defintion_formation
      := VIEW FROM (pdf : product_definition_formation)
              WHERE pdf.of_product :=: p;
END_VIEW;
```

An inline view can always be replaced with an explicit or partial binding to a named view.

12.4 Operations on extents

An extent is a set of bindings; as such, it may be used in expressions where a set is appropriate, and in particular an extent may be iterated over to visit each binding.

12.5 View expression evaluation

Given a binding, a view attribute expression may be evaluated. The result will be an instance in the underlying data set, a binding, or an extent.

12.6 FOR expression

The FOR expression is introduced for attribute assignment statements of MAP declarations to process a set of elements and to assign a set as a result to the target attribute. For this purpose, an iteration mechanism is used where all elements of the set can be processed step by step, selected, and manipulated.

The iteration of the FOR expression is controlled either by the repeat control known from EXPRESS (cf., ???). Alternatively, a more declarative approach can be specified using the FOR EACH concept. In the latter case, the following clauses are available.

– The EACH clause defines the (name of the) iterator variable. That is, in each processing step of the loop of the FOR expression, an element of the set is assigned to this iterator. The set is determined by the IN- (and the FROM-) clause.

– The IN clause specifies the set over which it has to be iterated over. This is either an (entity)

extend. In this case the FROM clause is optional. That is, if it shall be iterated over exact one (entity) extent without further restrictions the FROM clause need not to be specified. Alternatively, if it shall be iterated over an extent which is built upon many joined source extents, the FROM clause (and the WHERE clause) are needed.

In addition to the entity extent, it can also be iterated over an attribute of type AGGREGATE. In this case, the FROM clause is optional: if the source entity of this attribute to be iterated over is not specified in the FROM clause of the MAP declaration, it must be specified in the FROM clause of the FOR expression.

– The FROM clause of the FOR expression has the same semantics as the FROM clause of the MAP declaration (cf., ???).

– The WHERE clause of the FOR expression has the same semantics as the WHERE clause of the MAP declaration (cf., ???).

– The RETURN clause specifies an expression which has to be processed for each element during the iteration. All processed elements together build the result aggregate which is returned to the target attribute.

EXAMPLE 21 — FOR expression.

```
Source schema:
ENTITY product_definition;
  product_name : STRING;
  description : STRING;
END_ENTITY;

ENTITY product_definition_name;
  name    : STRING;
  of_product_definition : product_definition;
END_ENTITY;

Target schema:
ENTITY component;
  names : SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
```

In this example, the target entity `component` maps to the source entity `product_definition` and all instances of `product_definitio_name` which reference one instance of `product_definition` are grouped into the target attribute `component.names`. This is specified as follows.

```
Mapping definition:
MAP component
FROM pd : product_definition
SELECT
  description := pd.description;
  product_name := pd.product_name;
  names := FOR EACH pdn_instance
                IN pdn
             FROM pdn : product_definition_name
            WHERE pdn.of_product_definition :=: pd
           RETURN pdn_instance.name
END_MAP;
```

This example also shows that the scope of the FROM clause of the MAP declaration can be extended by the FROM clause of an FOR expression within this MAP declaration. That is, `product_definition_name` is not within the scope of the root entity of the FROM clause of the MAP declaration `product_definition`. In this case, the FOR expression specifies the so-called outer join operation. That is, for each instance of `product_definition` a target instance of `component` is built independent of the existence of instances of `product_definition_name` which references this `product_definition`. If such instances of `product_definition_name` do not exist, the value of `component.names` is the empty set. Otherwise, those instances (resp. the value `product_definition_name.name`) are assigned to the attribute `component.names`.

The RETURN clause can be nested in order to map attributes which are of type AGGREGATE OF AGGREGATE. This is shown in the following example.

EXAMPLE 22 — Nested FOR expression. The example 21 is extended as follows.

```
Source schema:
ENTITY product_definition;
  (* as defined in Ex. 21 *)
END_ENTITY;

ENTITY product_definition_name;
  (* as defined in Ex. 21 *)
END_ENTITY;

ENTITY product_definition_value;
  of_pdn : product_definition_name;
  value : STRING;
END_ENTITY;

Target schema:
ENTITY component;
  values : SET [0:?] OF SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
```

In addition to example 21, all instances of `product_definition_value` which reference one instance of `product_definition_name` are grouped together and are assigned to the inner aggregate of `component.values`. This is specified as follows.

```
Mapping definition:
MAP component
FROM pd : product_definition
SELECT
  description := pd.description;
  product_name := pd.product_name;
  names := FOR EACH pdn_instance
                IN pdn
              FROM pdn : product_definition_name
             WHERE pdn.of_product_definition :=: pd
            RETURN FOR EACH pdv_instance
                        IN pdv
                      FROM pdv : product_definition_value
                     WHERE pdv.of_pdn :=: pdn_instance
                    RETURN pdv_instance.value;
END_MAP;
```

The FOR expression also supports the so-called parallel iteration. That is, two or more iterator variables are assigned to elements of sets. During each step of the iteration loop, all the iterator variables are assigned to the next element of the corresponding set. This is shown in the following example.

EXAMPLE 23 — Parallel iteration with the FOR expression.

```
Source schema:
ENTITY persons;
  firstname : SET [0:?] OF STRING;
  lastname : SET [0:?] OF STRING;
END_ENTITY;
```

```
Target schema:
ENTITY set_of_persons;
  name : SET [0:?] OF STRING;
END_ENTITY;
```

It is assumed that `persons.firstname[i]` corresponds to `persons.lastname[i]` and that those two values have to be concatenated and have to be assigned to `set_of_persons.name[i]`.

```
Mapping specification:
MAP set_of_persons
FROM p : persons
SELECT
  name := FOR EACH firstname_value IN p.firstname AND
              EACH lastname_value  IN p.lastname
            RETURN firstname_value + lastname_value;
END_MAP;
```

This example also shows that the FROM clause of the FOR expression is optional when it is a subset of the FROM clause of the MAP declaration. In this example, no predicates are needed to select specific elements of the extent which is given by the IN clause. Thus, the WHERE clause is omitted.

Syntax:

```
50 for_expr = foreach_expr | forloop_expr .
51 foreach_expr = FOR EACH variable_id IN foreach_in_clause_arg { AND
   variable_id IN foreach_in_clause_arg } [ from_clause ] [ where_clause ]
   RETURN map_attr_assgnmt_expr ';' .
52 foreach_in_clause_arg = attribute_reference | view_attribute_reference
   | extent_reference .
54 forloop_expr = FOR repeat_control RETURN map_attr_assgnmt_expr ';' .
```

Rules and restrictions:

a) The target attribute of the attribute assignment statement where the FOR expression is used in must be of type AGGREGATE.

12.7 Conditional expression

This concept is introduced for MAP declarations so that a specified expression is assigned to a target attribute under some condition (or, else another expression is assigned). The conditional expressions can be nested.

Syntax:

```
73 map_cond_attr_expr = IF boolean_expression THEN map_attr_assgnmt_expr
   [ ELSE map_attr_assgnmt_expr ] END_IF ';' .
```

12.8 CASE expression

The CASE expression is similar to the CASE statement of EXPRESS.

EXAMPLE 24 — CASE expression.

```
MAP my_approval
FROM a : approval
SELECT
  status := CASE a.status OF
              'approved'     : 1;
              'not approved' : -1;
              'indetermined' : 0;
              OTHERWISE      : 2;
            END_CASE;
END_MAP;
```

Syntax:

```
40 case_expr = CASE selector OF { case_expr_action } [ OTHERWISE ':'
   expression ] END_CASE ';' .
41 case_expr_action = case_label { ',' case_label } ':' expression .
71 map_case_expr = CASE selector OF { map_case_expr_action } [ OTHERWISE
   ':' map_attr_assgnmt_expr ] END_CASE ';' .
72 map_case_expr_action = case_label { ',' case_label } ':'
   map_attr_assgnmt_expr .
```

13. Executable statements

EXPRESS-X has sixteen types of statements for use inside MAP declarations. Many of these statement types are taken directly from EXPRESS. Those that are either not in EXPRESS or are modified from their definition in EXPRESS include: the assignment statement, the FROM statement, the WHEN statement, the initialize statement, the DELETE statement, and the instantiation statement.

13.1 FOR clause

The FOR clause is used to control the instantiation of target instances. Without the FOR clause, for each qualified source instance or set of source instances given by the FROM and WHERE clause one target instance of each entity listed in the mapping header (target_entity_ref_list). The FOR clause allows to instantiate more than one target instance.

The FOR clause specifies a loop-control statement before the SELECT clause, i.e., before the attribute-assignment statements. The loop-control specifies the exact number of target instances that are built from each qualified source instance resp. set of qualified source instances. The attribute-assignment statements defines the mapping of all attributes of one or many target entities to some expressions.

Syntax:

```
45 entity_instantiation_loop = FOR instantiation_loop_control ';' .
65 instantiation_loop_control = instantiation_foreach_control |
   repeat_control .
64 instantiation_foreach_control = EACH variable_id IN
   source_attribute_reference  [ WITH_INDEX variable_id ]  { AND
   variable_id IN source_attribute_reference  [ WITH_INDEX variable_id ] }
   .
```

Rules and restrictions:

a) variable_id after the keyword EACH is of the same type as the elements of source_attribute_reference.

b) variable_id after the keyword INDEXING is of type NUMBER with values greater than one.

The loop control statement defines either an iterator over elements of a source attribute of type aggregate or it defines an iterator in a very similar way to the EXPRESS repeat loop.

In the first case, a so-called unnest operation will be applied. That is, the loop-control statement defines an iteration over a source attribute of type aggregate. In each iteration step, the next element of the source attribute is assigned to a variable and optionally the index position is assigned to a iterator variable. The value of this element can thus be used inside the FOR statement. For example, for each element of the source attribute of type aggregate a target instance can be generated and the element value can be assigned to a corresponding target attribute of type .

EXAMPLE 25 — In the following example, all item versions of one item are grouped together in the source data model. In contrast, each item version is a stand-alone instance in the target data model. This example shows that the FOR loop specifies an iteration over the elements of the source attribute `item_with_versions.id_of_versions`. For each source instance and for each element in that attribute a target instance is created. The target attribute `item_id` is mapped in the same way for all the target instances which of `item_version` which correspond to the same underlying `item_with_versions`. The target attribute `version_id` is assigned to the value of the iterator variable `version_iterator`.

```
ENTITY item_version; --target data model
  item_id    : STRING;
  version_id : STRING;
END_ENTITY;

ENTITY item_with_versions; -- source data model
  id             : STRING;
  id_of_versions : LIST OF STRING;
END_ENTITY;
```

```
MAP iv : LIST [0:?] OF item_version
FROM iwv : item_with_versions;
FOR EACH version_iterator OF iwv.id_of_versions INDEXING i
SELECT
  iv[i].item_id    := iwv.id;
  iv[i].version_id := version_iterator;
END_MAP;
```

For example, the following target instances are built from the source instance below.
Source instance set:

```
#1 = item_with_versions(1,(10,11,12));
```

Target instance set:

```
#1 = item_version(1,10);
#2 = item_version(1,11);
#3 = item_version(1,12);
```

Alternatively, the repeat-loop control statement known from EXPRESS can be used to specify the iteration steps.

EXAMPLE 26 — In the following example, all item versions of one item are grouped together in the source data model. In contrast, each item version is a stand-alone instance in the target data model. This example shows that the FOR loop specifies an iteration over the elements of the source attribute `item_with_versions.id_of_versions`. For each source instance and for each element in that attribute a target instance is created. The target attribute `item_id` is mapped in the same way for all the target instances which of `item_version` which correspond to the same underlying `item_with_versions`. The target attribute `version_id` is assigned to the value of the iterator variable `version_iterator`.

```
SCHEMA target;
ENTITY parent;
END_ENTITY;

ENTITY child;
  parent : parent;
END_ENTITY;
END_SCHEMA; -- target
```

```
SCHEMA source;
ENTITY parent;
  children : INTEGER;
END_ENTITY;

END_SCHEMA; -- source
```

```
MAP tp : parent
FROM sp : parent
END_MAP;
```

```
MAP c : LIST [0:?] OF child
FROM p : parent
FOR i := 1 TO p.children
SELECT
  parent := parent();
END_MAP;
```

Alternatively, one single MAP can be specified for this example as shown below.

```
MAP  tp : parent, c : LIST [0:?] OF child
NAMED   group_parent_and_child
FROM sp: parent
FOR i := 1 TO p.children
SELECT
  parent := parent@group_parent_and_child();
END_MAP;
```

This statement can only be used within a MAP declaration.

14. Built-in functions and procedures

15. Execution model semantics

The execution model semantics of EXPRESS-X are described according to the main concepts of the language. The following source schema and corresponding source instance set will be used throughout this clause.

EXAMPLE 27 — .

```
SCHEMA EXAMPLE_SCHEMA;

ENTITY item;
  id : STRING;
  its_version : item_version;
  approved_by : STRING;
END_ENTITY;

ENTITY item_version;
  id : STRING;
  its_ddid : OPTIONAL ddid;
END_ENTITY;

ENTITY ddid;
  id : STRING;
END_ENTITY;

ENTITY person;
  name : STRING;
END_ENTITY;
```

```
END_SCHEMA;


#1 = item('i_1',#3,'smith');
#2 = item('i_2',#4,'jones');
#3 = item_version('iv_1',#5);
#4 = item_version('iv_2');
#5 = ddid('ddid_1');
#6 = person('smith');
#7 = person('jones');
#8 = person('miller');
```

15.1 Reference of source (and target) schemas

During the execution of the EXPRESS-X specification schema, the underlying source schemas (and the underlying target schemas in case of the SCHEMA_MAP) together with their type definitions are made available.

15.2 Inclusion of externally defined functions

The referenced functions which are defined externally to the EXPRESS-X specification have to be available for execution during runtime.

NOTE — This concept is only available in a SCHEMA_MAP declaration.

15.3 Import of mappings

All definitions and declarations which are specified in the referenced SCHEMA_MAP are available during the execution of the EXPRESS-X specification. They are handled as if they were specified inside the referencing EXPRESS-X specification. That is, they shall be considered not as underlying but as additional definitions and declarations.

NOTE — This concept is only available in a SCHEMA_MAP declaration.

15.4 Type mapping

The attribute assignments are identified which map between source and target attributes of data types which are mapped in the TYPE_MAP declaration. The corresponding type mapping expression is then used as the cast operation for that source attribute in the assignment. The same is true for the computation of the inverse mappings if the mapping implementation supports this conformance class.

EXAMPLE 28 — In Ex. 15 the target type `dmark` is mapped to the source type `dollar` by multiplying `dollar` with the factor `1.5` to derive `dmark`. Any attribute assignment where a target attribute of type `dmark` is mapped to some attributes where at least one of them is of type `dollar`, the expression of the TYPE_MAP is first applied to the(se) source attribute(s). That is, they are first multiplied with the factor `1.5`.

NOTE — This concept is only available in a SCHEMA_MAP declaration.

15.5 The FROM clause

The FROM clause specifies the scope of the VIEW / MAP declaration. That is, the cartesian product of all entities which are listed in the FROM clause (including those entities which are directly or indirectly referenced by them) build the basis for further processing. Thus, the result of processing of the FROM clause is an input data stream where instances of all specified entities are merged together by the cartesian product. The execution model semantics will be detailed using the following example.

EXAMPLE 29 — A view is built over two (root) entities (the entity `item_version` is directly referenced by the root entity `item` and `ddid` is indirectly referenced).

```
VIEW items_and_persons
FROM item, person
END_VIEW;
```

NOTE — It shall be emphasized that the execution model semantics which are described below are valid for the execution of a VIEW as well as a MAP declaration. Thus, the VIEW declaration of Ex. 29 can be replaced by a corresponding MAP declaration and a corresponding target schema.

During runtime, the output stream of the view is built from the following entities:

- all entities which are specified in the FROM clause (i.e., `item` and product in Ex. 27)
- all entities which are directly referenced by the FROM-clause entities (i.e., `item_version` which is referenced by `item`) and
- all entities which are indirectly referenced by the FROM-clause entities (i.e., `ddid` which is referenced by `item_version` which is itself referenced by `item`).

The cartesian product is built over all entities which are explicitly specified in the FROM clause. Consequently, the data of the directly and indirectly referenced entities is implicitly part of the output stream.

EXAMPLE 30 — After the processing of the FROM clause, the source data set as specified in Ex. 27 is represented in the following way in the output stream. (0x... are used as internal IDs for the instances)

	item				item_version			ddid		person	
		id	its_version	approved_by		id	its_ddid		id		name
0x01	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#6	smith
0x02	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#7	jones
0x03	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#8	miller
0x04	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#6	smith
0x05	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#7	jones
0x06	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#8	miller

It shall be emphasized that only those combinations of `item` and `item_version` are part of the output stream which are connected by relationships via `item.its_version`.

15.6 The WHERE clause

The WHERE clause specifies predicates all instances of the output stream have to fullfil. That is, according to those predicates the output stream is filtered.

EXAMPLE 31 — The following example extends the VIEW declaration of Ex. 29 by an WHERE clause to filter specific persons and to join items and persons.

```
VIEW items_and_persons
FROM i : item, p : person
WHERE (p.name = 'smith' OR p.name = 'jones') AND
      i.approved_by = p.name
END_VIEW;
```

NOTE — It shall be emphasized that the execution model semantics which are described below are valid for the execution of a VIEW as well as a MAP declaration. Thus, the VIEW declaration of Ex. 31 can be replaced by a corresponding MAP declaration and a corresponding target schema.

After the evaluation of the WHERE clause predicates, the output stream will be modified as follows: all grey boxes will be filtered out.

	item				item_version			ddid		person	
		id	its_version	approved_by		id	its_ddid		id		name
0x01	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#6	smith
0x02	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#7	jones
0x03	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#8	miller
0x04	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#6	smith
0x05	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#7	jones
0x06	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#8	miller

15.7 The IDENTIFIED_BY clause

The IDENTIFIED_BY clause has effects during the execution of the mapping specification.

– Assignment of internal IDs to the instances of the output stream.

– Uniqueness according to the IDENTIFIED_BY expression has to be ensured.

EXAMPLE 32 — For the explanation of the execution model semantics of the IDENTIFIED_BY clause another source schema, source instance set and view declaration is used.

```
ENTITY person;
  first_name : STRING;
  last_name  : STRING;
END_ENTITY;
```

```
#1 = person('marc','jones');
#2 = person('paul','jones');
#3 = person('paul','smith');
```

```
VIEW view_persons
FROM p : person
IDENTIFIED_BY p.last_name;
SELECT
  name : STRING := p.last_name;
END_VIEW;
```

The result of evaluating the IDENTIFIED_BY clause are the following view instances (grey boxes will be filtered, black boxes are added).

view terms	view_person				
		internal ID	corresponding source IDs		name
source terms	person				
				first_name	last_name
0x01	#1	jones	{#1,#2}	marc	jones
0x02	#2			paul	jones
0x03	#3	smith	#3	paul	smith

NOTE — It shall be emphasized that the execution model semantics which are described below are valid for the execution of a VIEW as well as a MAP declaration. Thus, the VIEW declaration of Ex. 32 can be replaced by a corresponding MAP declaration and a corresponding target schema.

15.8 The SELECT clause

The SELECT clause and the subsumed attribute assignments project attributes and/or entities out of the output stream.

EXAMPLE 33 — In this example, the execution model semantics are explained when only one entity is specified in the SELECT clause.

```
VIEW items_and_persons
FROM i : item, p : person
SELECT i;
END_VIEW;
```

After the evaluation of the SELECT clause, the output stream will be modified as follows: all grey boxes will be filtered out.

	item				item_version			ddid		person	
		id	its_version	approved_by		id	its_ddid		id		name
0x01	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#6	smith
0x02	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#7	jones
0x03	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#8	miller
0x04	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#6	smith
0x05	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#7	jones
0x06	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#8	miller

Alternatively, if some attribute assignments are specified also attributes and/or entities are projected. In the following example, no additional expressions are specified in the attribute assign-

mens in order to separate the different execution model semantics.

```
VIEW items_and_persons
FROM i : item, p : person
SELECT

END_VIEW;
```

NOTE — It shall be emphasized that the execution model semantics which are described below are valid for the execution of a VIEW as well as a MAP declaration. Thus, the VIEW declaration of Ex. 33 and Ex. 33 can be replaced by a corresponding MAP declaration and a corresponding target schema.

15.9 Partitions

???

15.10 Network mapping

???

15.11 The FOR statement

The FOR statements specifies a loop so that for each instance of the output stream this loop is executed as many times as specified by the loop control statement (cf., Sect. 13.1).

EXAMPLE 34 — We assume that for each source instance of item exactly three corresponding target instances have to be generated. That is specified in the following mapping specification.

```
ENTITY item_with_duplicates;
  id : STRING;
  index : INTEGER;
END_ENTITY;
```

```
MAP iwd : LIST [3:3] OF item_with_duplicates
FROM i : item
SELECT
  FOR var := 1 TO 3
    id := i.id;
    index := var;
  END_FOR;
END_MAP;
```

	item_with_duplicates									
		id			index					
	item					item_version			ddid	
		id	its_version	approved_by			id	its_ddid		id
0x01	#1	i_1	#3	smith	1	#3	iv_1	#5	#5	ddid_1
0x02	#1	i_1	#3	smith	2	#3	iv_1	#5	#5	ddid_1
0x03	#1	i_1	#3	smith	3	#3	iv_1	#5	#5	ddid_1
0x04	#2	i_2	#4	jones	1	#4	iv_2		#5	ddid_1
0x05	#2	i_2	#4	jones	2	#4	iv_2		#5	ddid_1
0x06	#2	i_2	#4	jones	3	#4	iv_2		#5	ddid_1

NOTE — This concept is only available in a SCHEMA_MAP declaration.

15.12 Explicit binding

???

Annex A
(normative)
EXPRESS-X language syntax

This annex defines the lexical elements of the language and the grammar rules that these elements shall obey.

NOTE — This syntax definition will result in ambiguous parsers if used directly. It has been written so as to convey information regarding the use of identifiers. The interpreted identifiers define tokens that are references to declared identifiers, and therefore should not resolve to simple_id. This requires a parser developer to enable identifier reference resolution and return the required reference token to a grammar rule checker.

All of the grammar rules of EXPRESS specified in annex A of ISO 10303-11:1994 are also grammar rules of EXPRESS-X. In addition, the grammar rules specified in the remainder of this annex are grammar rules of EXPRESS-X.

A.1 Tokens

The following rules specify the tokens used in EXPRESS-X. Except where explicitly stated in the syntax rules, no white space or remarks shall appear within the text matched by a single syntax rule in the following clauses.

A.1.1 Keywords

This subclause gives the rules used to represent the keywords of EXPRESS-X.

NOTE — This subclause follows the typographical convention that each keyword is represented by a syntax rule whose left hand side is that keyword in uppercase.

NOTE — All the keywords of EXPRESS are also keywords of EXPRESS-X

```
 1  CREATE = 'create'.
 2  EACH = 'each'.
 3  END_CREATE = 'end_create'.
 4  END_EXTERNAL = 'end_external'.
 5  END_FOR = 'end_for'.
 6  END_INLINE_FUNCTION = 'end_inline_function'.
 7  END_MAP = 'end_map'.
 8  END_SCHEMA_MAP = 'end_schema_map'.
 9  END_SCHEMA_VIEW = 'end_schema_view'.
10  END_TYPE_MAP = 'end_type_map'.
11  END_VIEW = 'end_view'.
12  EXTERNAL = 'external'.
13  IDENTIFIED_BY = 'identified_by'.
14  IMPORT_MAPPING = 'import_mapping'.
```

```
15  INLINE_FUNCTION = 'inline_function'.
16  INSTANCE_OF = 'instance_of'.
17  MAP = 'map'.
18  NAMED = 'named'.
19  PARTITION = 'partition'.
20  SCHEMA_MAP = 'schema_map'.
21  SCHEMA_VIEW = ’schema_view’.
22  SOURCE = 'source'.
23  TARGET = 'target'.
24  TYPE_MAP = 'type_map'.
25  VIEW = 'view'.
```

A.1.2 Character classes

```
26  digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
27  letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
           | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
           | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .
28  simple_id = letter { letter  | digit  | '_' } .
```

A.1.3 Interpreted identifiers

NOTE — All interpreted identifiers of EXPRESS are also interpreted in EXPRESS-X

```
29  instance_ref = instance_id .
30  network_ref = network_id .
31  partition_ref = partition_id .
32  schema_map_ref = schema_map_id .
33  schema_view_ref = schema_view_id .
34  source_schema_ref = schema_ref .
35  target_schema_ref = schema_ref .
36  view_attribute_ref = view_attribute_id .
37  view_ref = view_id .
```

A.2 Grammar rules

```
38  attribute_reference =   attribute_ref
    | primary_extended attribute_qualifier  .

39  boolean_expression = expression .

40  case_expr = CASE selector OF { case_expr_action }
    [ OTHERWISE ':' expression ]   END_CASE ';' .

41  case_expr_action =   case_label { ',' case_label } ':' expression .
```

```
42 complex_entity_spec = entity_reference AND entity_reference { AND
   entity_reference }.
43 create_map_decl = CREATE instance_id INSTANCE_OF
   target_entity_reference ';' map_attr_decl_stmt_list  END_CREATE ';'  .
44 create_view_decl =  CREATE instance_id INSTANCE_OF VIEW view_reference
   ';' view_attr_decl_stmt_list  END_CREATE ';'  .
45 entity_instantiation_loop = FOR instantiation_loop_control ';' .
46 entity_qualifier = '.' entity_ref .
47 entity_reference = entity_ref | primary_extended entity_qualifier .
48 extent_reference = source_entity_reference | view_reference .
49 external_functions_spec = EXTERNAL function_head { function_head }
                            END_EXTERNAL ';' .
50 for_expr = foreach_expr | forloop_expr .
51 foreach_expr = FOR EACH variable_id IN foreach_in_clause_arg
                    { AND variable_id IN foreach_in_clause_arg }
                  [ from_clause ] [ where_clause ]
                  RETURN map_attr_assgnmt_expr ';' .
52 foreach_in_clause_arg = attribute_reference | view_attribute_reference
   | extent_reference .
53 foreign_ref = schema_ref | schema_view_ref | schema_map_ref .
54 forloop_expr = FOR repeat_control RETURN map_attr_assgnmt_expr ';' .
55 from_clause = FROM ( '(' from_parameter_list ')'
                     | from_parameter_list ) .
56 from_parameter = [ parameter_id { parameter_id } ':' ] extent_reference
   .
57 from_parameter_list = from_parameter { ',' from_parameter } .
58 identified_by_clause = IDENTIFIED_BY expression { ',' expression } ';'.
59 inline_funct_head = INLINE_FUNCTION [ '(' formal_parameter { ';'
   formal_parameter } ')' ] ':' parameter_type ';'  .
60 inline_function_decl = inline_funct_head [ algorithm_head ]
                         stmt { stmt } END_INLINE_FUNCTION ';' .
61 inline_view_decl = VIEW from_clause [ where_clause ]
                    [ view_project_clause ] END_VIEW ';' .
62 instance_id = simple_id .
63 instance_qualifier = '.' instance_ref .
64 instantiation_foreach_control = EACH variable_id
                                  IN source_attribute_reference
                                  [ WITH_INDEX variable_id ]
                                  { AND variable_id
                                  IN source_attribute_reference
                                  [ WITH_INDEX variable_id ] }  .
65 instantiation_loop_control = instantiation_foreach_control
                             | repeat_control .
66 map_attr_assgnmt_expr =   expression  | map_cond_attr_expr  |
   map_case_expr | for_expr  | inline_function_decl  | map_call  .
67 map_attr_decl_stmt_list = map_attribute_declaration
                            { map_attribute_declaration } .
```

```
68  map_attribute_decl_block = map_attr_decl_stmt_list .
69  map_attribute_declaration =   [ entity_reference '.' ] attribute_ref
    ':=' map_attr_assgnmt_expr ';'  .
70  map_call = entity_reference [ '@' network_or_partition_qualification ]
                          '(' expression { ',' expression } ')' .
71  map_case_expr = CASE selector OF { map_case_expr_action }
    [ OTHERWISE ':' map_attr_assgnmt_expr ]   END_CASE ';' .
72  map_case_expr_action = case_label { ',' case_label } ':'
    map_attr_assgnmt_expr .
73  map_cond_attr_expr = IF boolean_expression THEN map_attr_assgnmt_expr
                         [ ELSE map_attr_assgnmt_expr ] END_IF ';' .
74  map_decl = MAP map_decl_header
               ( (map_decl_body { map_partitions }) | map_decl_body )
               END_MAP ';' .
75  map_decl_body = ( ( from_clause [ identified_by_clause ] ) |
                      subtype_of_clause )
                    [ where_clause ]
                    { entity_instantiation_loop }
                    [ map_project_clause ] .
76  map_decl_header = target_entity_ref_list [ NAMED network_id ] .
77  map_interface_spec = IMPORT_MAPPING  schema_map_or_view_ref_or_rename
                         { ',' schema_map_or_view_ref_or_rename } ’;’.
78  map_partition =  PARTITION [ partition_id ':' ] map_decl_body .
79  map_partitions = map_partition { map_partition } .
80  map_project_clause = SELECT ( extent_reference
                       | map_attribute_decl_block ).
81  network_id = simple_id .
82  network_or_partition_qualification = network_ref
    | [ network_ref '.'] partition_ref .
83  partition_id = simple_id .
84  primary_extended = qualifiable_factor_extended { qualifier_extended }
    .
85  qualifiable_factor_extended =   qualifiable_factor  | schema_map_ref  |
    schema_view_ref  | view_ref  | map_call  | view_call  |
    view_attribute_ref  | instance_ref  .
86  qualifier_extended =   qualifier  | view_qualifier  |
    instance_qualifier  | entity_qualifier  | view_attribute_qualifier  .
87  reference_clause = REFERENCE FROM foreign_ref
                       [ '(' resource_or_rename
                             { ',' resource_or_rename } ')' ] ';' .
88  schema_alias_id = schema_id .
89  schema_map_alias_id = schema_map_id .
90  schema_map_body_element = function_decl | procedure_decl | view_decl
                            | create_view_decl | map_decl
                            | create_map_decl .
91  schema_map_body_element_list = schema_map_body_element
                                   { schema_map_body_element } .
```

```
 92  schema_map_decl = SCHEMA_MAP schema_map_id
                       target_interface_spec source_interface_spec
                       { map_interface_spec } { external_functions_spec }
                       { type_mapping_stmt } [ constant_decl ]
                       schema_map_body_element_list END_SCHEMA_MAP ';' .
 93  schema_map_id =  simple_id .
 94  schema_map_or_view_ref_or_rename = schema_map_ref_or_rename
                                      | schema_view_ref_or_rename .
 95  schema_map_ref_or_rename =  [ schema_map_alias_id ':' ]
                                 schema_map_ref .
 96  schema_ref_or_rename = [ schema_alias_id ':' ] schema_ref .
 97  schema_view_alias_id = schema_view_id .
 98  schema_view_body_element = function_decl | procedure_decl | view_decl
                              | create_view_decl .
 99  schema_view_body_element_list =   schema_view_body_element   {
     schema_view_body_element }  .
100  schema_view_decl = SCHEMA_VIEW schema_view_id { reference_clause }
                        [ constant_decl ]
                        schema_view_body_element_list END_SCHEMA_VIEW ';' .
101  schema_view_id = simple_id .
102  schema_view_ref_or_rename = [ schema_view_alias_id ':' ]
                                 schema_view_ref .
103  source_attribute_reference = attribute_reference  |
     view_attribute_reference .
104  source_entity_reference = entity_reference .
105  source_interface_spec = SOURCE schema_ref_or_rename
                             { ',' schema_ref_or_rename } ';' .
106  source_type_reference = type_reference .
107  subtype_of_clause =  SUBTYPE OF view_or_entity_reference
                          [ PARTITION partition_ref ] ';' .
108  syntax = schema_map_decl | schema_view_decl .
109  target_entity_alias_id =   entity_id  .
110  target_entity_ref_list =  target_entity_ref_list_el
                               { ',' target_entity_ref_list_el } .
111  target_entity_ref_list_el =  [ target_entity_alias_id
                                    ':' [ LIST bound_spec OF ] ]
                                    target_entity_reference .
112  target_entity_reference =   entity_reference  | complex_entity_spec  |
     target_schema_ref '.' '(' complex_entity_spec ')'  .
113  target_interface_spec = TARGET schema_ref_or_rename
                             { ',' schema_ref_or_rename } ';' .
114  target_type_reference = type_reference .
115  type_assgnmt_expr =   expression | case_expr .
116  type_map_stmt_body = [ schema_ref '.' ] base_type ':='
     type_assgnmt_expr ';'.
```

```
117 type_mapping_stmt =   TYPE_MAP target_type_reference
    { target_type_reference }
    FROM source_type_reference { source_type_reference } ';'
    { type_map_stmt_body }  END_TYPE_MAP ';' .
118 type_reference = [ schema_ref '.' ] type_ref .
119 view_attr_assgnmt_expr = expression | view_cond_attr_expr |
    view_case_expr | inline_view_decl | view_call .
120 view_attr_decl_stmt_list = view_attribute_decl
                              { view_attribute_decl } .
121 view_attribute_decl = view_attribute_id ':' [ source_schema_ref '.' ]
                          base_type ':=' view_attr_assgnmt_expr ';' .
122 view_attribute_id = simple_id .
123 view_attribute_qualifier = '.' view_attribute_ref .
124 view_attribute_reference = view_attribute_ref
    | primary_extended view_attribute_qualifier  .
125 view_call = view_reference
              '(' view_call_argument { ',' view_call_argument } ')' .
126 view_call_argument = expression | view_call .
127 view_case_expr = CASE selector OF { view_case_expr_action }
    [ OTHERWISE ':' view_attr_assgnmt_expr ]   END_CASE ';' .
128 view_case_expr_action = case_label { ',' case_label } ':'
    view_attr_assgnmt_expr .
129 view_cond_attr_expr = IF boolean_expression
                          THEN view_attr_assgnmt_expr
                          [ ELSE view_attr_assgnmt_expr ] END_IF ';' .
130 view_decl = VIEW view_id ';' [ subtype_of_clause ]
              ( view_partitions  | view_decl_body )
              END_VIEW ';' .
131 view_decl_body =  [ from_clause ] [ identified_by_clause ]
                     [ where_clause ] [ view_project_clause ] .
132 view_id = simple_id .
133 view_or_entity_reference =  view_reference | entity_reference .
134 view_partition =   PARTITION [ partition_id ':' ] view_decl_body .
135 view_partitions = view_partition { view_partition } .
136 view_project_clause =  SELECT (extent_reference
                        | view_attr_decl_stmt_list) .
137 view_qualifier = ’.’ view_ref .
138 view_reference = view_ref | primary_extended ’.’ view_qualifier .
```

A.3 EXPRESS Syntax

```
139 add_like_op = '+'| '-' | OR | XOR .
140 bound_1 = numeric_expression .
141 bound_2 = numeric_expression .
142 bound_spec = '[' bound_1 ':' bound_2 ']' .
143 built_in_constant = CONST_E | PI | SELF | '?' .
```

```
144  built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS
                       | EXP | FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND
                       | LOINDEX | LOG | LOG2 | LOG10 | NVL | ODD | ROLESOF
                       | SIN | SIZEOF | SQRT | TAN | TYPEOF | USEDIN | VALUE
                       | VALUE_IN | VALUE_UNIQUE .
145  constant_factor = built_in_constant | constant_ref .
146  enumeration_reference = [ type_ref '.' ] enumeration_ref .
147  expression = simple_expression [ rel_op_extended simple_expression ] .
148  factor = simple_factor [ '**' simple_factor ] .
149  logical_expression =  expression .
150  numeric_expression = simple_expression .
151  repeat_control = [ increment_control ] [ while_control ]
                    [ until_control ] .
152  simple_factor = aggregate_initializer | entity_constructor
                  | enumeration_reference | interval | query_expression
                  | ( [ unary_op ] ( '(' expression ')'  | primary ) ) .
```

A.4 Cross reference listing

Annex B
(informative)
Bibliography

EXPRESS-V language (ISO TC184/SC4/WG5 N251).

EXPRESS-M language (ISO TC184/SC4/WG5 N243).

BRITTY language.

Wirth, Niklaus, *"What can we do about the unnecessary diversity of notations for syntactic definitions?*," Communications of the ACM, November 1977, v. 20, no. 11, p. 822.